# 1 模板的定义

|  |  |
| --- | --- |
| //函数模板 template<模板参数列表> 函数定义体 | //类模板 template<模板参数列表> 类定义体 |

## 1.1 模板参数列表

        模板参数列表是一个用尖括号（< >）包围的和用逗号分隔若干个模板形式参数的列表，且在模板定义中，模板形式参数列表不能为空：

|  |
| --- |
| //函数模板定义template<typename T>void  DemoFunc1(T param){    ...}template< >void  DemoFunc2(T param){ //编译报错，因为模板参数列表不能为空！    ...}//类模板定义template<typename T>class DemoClass1{};template< >class DemoClass2{   //编译报错，因为模板参数列表不能为空！}; |

## 1.2 模板形参

### 1.2.1 模板类型参数

        一般来说，我们可以将模板类型参数看作类型说明符（如int、string、auto、decltype等）来使用，因此模板类型参数可用来指定函数返回类型、函数的形参类型、变量的声明类型或指定类型转换时需要转换的目标类型，如下所示：

|  |
| --- |
| //T1, T2, T3和T4分别用于函数返回类型、函数的形参类型、变量的声明类型和指定类型转换时需要转换的目标类型template<class T1, class T2, class T3, class T4>T1  func(T2  param){    T3 var1 = T3( );    T4 var2 = static\_cast<T4>(var1);} |

1.2.1.1 class和typename

        模板形式参数列表中的每一个类型参数前都必须使用关键字class或typename，且在模板形式参数列表中，这两个关键字的含义相同，可以互换，在同一个模板形式参数列表中甚至可以同时使用这两个关键字，如下所示：

|  |
| --- |
| template<typename T, class U>class DemoClass{}; |

        需要注意的是，关键字class比关键字typename更早应用于模板参数列表中，而另一方面，当使用class指定模板类型参数时，并不代表该模板类型参数的实参只能是类类型，它也可以是非类类型（也即内置类型），由此可见，从名字的角度，typename比class更直观。

### 1.2.2 模板非类型参数（模板值参数）

        在模板定义内，模板非类型参数是一个常量值，在需要常量表达式的地方，都可以使用模板非类型参数，例如，使用模板非类型参数指定数组大小，如标准库中的std::array：

|  |
| --- |
| template < class T, size\_t N >class array;std::array<int, 10> arr;  //定义一个元素类型为int的包含10个元素的数组 |

### 1.2.3 模板形参名字的作用域

        一个模板形式参数名的可用范围是在其声明之后，直至模板声明或定义结束之前（遵循通用的标识符作用域规则）；

        模板形式参数名也会隐藏外层作用域中声明的相同名字（也与任何其他名字可隐藏外层作用域的相同名字的规则相同）；

        但是，在模板内不能重用模板形式参数名称（也即在模板内部不能声明与模板形参同名的名字），而由于模板形式参数不能被重用，所以一个模板参数名在一个特定模板参数列表中也只能出现一次。

|  |
| --- |
| typedef double A;  template<class A, class B>    //模板形参名字"A"会隐藏在外层作用域定义的类型定义名称"A"  void  func(A  a, B  b){      A  tmp = a;     //变量tmp的类型是模板参数A的类型，而非double      double  B;      //编译报错，因为在模板内部不能重用模板参数名称"B"  }  template<class  V, class  V>  //编译报错，因为模板形参名字不能重复  void  func(V  a){} |

### 1.2.4 使用模板类型参数的类型成员

        假定T是一个模板类型参数，在编译模板本身的阶段，当编译器遇到类似"T::mem"这样的代码时，它不会知道mem是一个类型成员还是一个static数据成员，直至模板实例化阶段时才会知道。

        但是，为了处理模板，编译器必须在编译模板本身时知道名字是否表示一个类型（以此来判断模板中的各个语句是否合法），因此在默认情况下，C++语言假定通过模板类型参数和作用域运算符（::）访问的名字不是类型，而是static数据成员；而当我们希望通知编译器一个通过模板类型参数和作用域运算符（::）访问的名字表示类型时，必须使用关键字typename进行修饰。

|  |
| --- |
| template<class  It>typename It::value\_type  func(It  beg, It  end) {    //处理序列    return  \*beg;   //返回序列中一个元素的拷贝或引用} |

### 1.2.5 模板形参的默认实参值

        就像我们能为函数参数提供默认实参一样，我们也可以为模板形参提供默认模板实参。在C++11标准中，我们可以同时为函数模板和类模板的模板形参提供默认实参。另外，和函数默认实参一样，对于一个模板参数，只有当它右侧的所有模板形参都有默认实参值时，它才可以由默认实参值。

1.2.5.1 函数模板默认实参值

|  |
| --- |
| /\*    我们为下面的函数模板提供了模板默认实参，也为其相应的模板函数提供了函数默认实参。    默认模板实参指出模板形参F将使用标准库的less函数对象类，且它是使用与compare一样的模板类型参数T进行实例化的；    而函数默认实参则指出函数形参f将是类型F的一个默认初始化的对象\*/template<typename T, typename F=less<T> /\*模板默认实参\*/ >bool compare(const T &v1, const T &v2, F f=F() /\*函数默认实参\*/ ){    return f(v1, v2);} |

1.2.5.2 类模板默认实参值

        无论何时（不管模板形参是否具有默认实参值），我们在使用类模板时，都必须在类模板名之后接上尖括号，该尖括号用于指出所使用的类必须从一个类模板实例化而来，如下所示：

|  |
| --- |
| //类模板定义template<typename T = int>class DemoClass1{};void  main(void){    DemoClass1< > demo;   //等价于"DemoClass1<int> demo;"，对于类模板不能省略尖括号} |

# 2 模板的声明

        模板的声明必须包含模板形式参数列表，且声明中的模板参数的名字以及函数形参的名字不必与定义中的相同，以及模板的声明必须和其定义有相同数量和种类（即模板类型参数和模板非类型参数）的模板参数，如下所示：

|  |
| --- |
| //函数模板声明template<class  T>int  compare(const T &, const T &); //函数模板声明中的函数形参名字可以省略//函数模板定义template<class  U>int  compare(const U &a, const U &b){ /\* ... \*/ }//类模板声明template<class  T>class  Blob;//类模板定义template<class  U>class  Blob{ /\* ... \*/ };//因为类模板声明中不需使用到模板形式参数，所以我们还可以省略模板形式参数的名字//类模板声明template<class>       //此处省略了模板参数的名字class  Blob; |

        需要注意的是，在声明派生类模板时，不需要包含继承关系，如下所示：

|  |
| --- |
| //类模板定义template<class  T>class  Demo: std::vector<T>{    ...};//类模板声明template<class>class  Demo; |

# 3 模板的编译和组织

当编译器遇到一个模板定义时，它并不生成代码，只有当我们实例化出模板的一个特定版本时（即当我们使用模板时），编译器才会生成代码。这一特性影响了模板提供者和模板使用者应该如何正确地组织代码，以及编译器何时才检测到模板编译错误。

## 3.1 模板编译三阶段

        模板直到实例化时才会生成代码，这一特性影响了我们何时才会获知定义模板的代码或者使用模板的代码的编译错误。通常，编译器会在以下三个阶段报告错误：

（1）编译模板本身

（2）遇到模板的使用

（3）模板实例化

        需要注意的是，并不是所有模板都会被编译！因为编译器只会对源文件进行编译，不会对头文件进行编译，而模板通常都是定义在头文件中的，所以如果该模板所属的头文件没有被任何源文件包含（在源文件中包含头文件就是将头文件的代码在源文件中展开，使得该源文件包含了头文件中的所有代码），那么编译器不会对模板定义的代码进行编译，此时即使模板定义存在语法错误，编译器也不会报错。因此，当我们在设计模板代码时，为了尽早发现模板定义中的错误，应在某个源文件中包含该模板所属的头文件。

### 3.1.1 编译模板本身

        在这个阶段，编译器通常不会发现很多错误，编译器可以检查语法错误（如忘记分号或者变量名拼错等，但也就这么多了）。如下所示（假设模板所属的头文件已被某个源文件所包含）：

|  |
| --- |
| //DemoA.h#ifndef DEMOA\_H#define DEMOA\_Htemplate<class T>class DemoA{public:    DemoA(){}    DemoA(const T &val) : tt(val)       //编译报错，变量名拼写错误（将t误写成tt）    {}private:    T  t    //编译报错，缺少分号}；template<class T>void f(T a){    //在默认在情况下，编译器认为"T::a"表示静态成员变量名称，    //显然不能使用变量名称定义局部变量y1，因此这里发生语法错误    T::a y1;    //编译报错    typename T::a y2;   //编译通过，使用typename关键字显式指定"T::a"表示类型名}#endif |

### 3.1.2 遇到模板的使用

        当编译器遇到使用模板的代码时，仍然不会进行太多的检查：对于函数模板的调用，编译器通常会检查函数实参数目/类型是否和函数形参数目/类型相一致；对于类模板的调用，编译器可以检查用户是否提供了正确数目的模板实参，但也仅限于此了。如下所示：

|  |
| --- |
| //DemoA.h#ifndef A\_H#define A\_Htemplate<class T>class DemoA{public:    DemoA(){}    DemoA(const T &val):t(val){}private:    T  t;};#endif |

|  |
| --- |
| //DemoB.h#ifndef B\_H#define B\_Htemplate<class T>T DemoBFunc(const T &t){    return t;}#endif |

|  |
| --- |
| //main.cpp#include "DemoA.h"#include "DemoB.h"int main(void){    //编译报错：no matching function for call to ‘DemoBFunc()’    DemoBFunc();    //原因：函数实参数目少于所需的形参数目而导致编译器找不到可行函数    //编译报错：wrong number of template arguments (2, should be 1)    DemoA<int, int> objA; //原因：实参数目和所需形参数目不一致    return 0;} |

### 3.1.3 模板实例化

        当编译器实例化一个模板时，它使用模板实际参数代替对应的模板形式参数来创建出模板的一个实例。只有在这个阶段才能发现类型相关的错误（如若模板定义假设模板类型参数支持<运算符（即在模板定义中使用了与模板类型参数相关的<运算符操作），但我们传递给模板类型参数的实参类型并不支持<运算符，那么编译器会在模板实例化时报错），这依赖于编译器如何管理实例化，而且这类错误可能在链接时才报告。

        需要注意的是，当我们编写模板定义时，代码不能是针对特定类型的（即不针对特定模板类型实参），但模板代码通常可以对其所使用的类型作一些假设：如假设模板类型参数支持<运算符、支持自增++或自减--运算符、支持默认构造函数等。因此模板的使用者必须保证传递给模板的实参支持模板所要求的操作，以及这些操作在模板中能正确工作。（模板编程假设）

        注：大多数模板编译错误都是在实例化期间报告的！

3.1.3.1 模板实例化（在使用时才实例化）

        在默认情况下，一个函数模板或类模板的成员函数（包括类模板的构造函数和静态成员函数）只有当程序用到它时才会进行实例化。特别地，如果一个类模板的成员函数没有被使用，则它不会被实例化，这一特性使得即使某种类型不能完全符合模板操作的要求，我们仍然能用该类型实例化类，如下所示：

|  |
| --- |
| // DemoA.h#ifndef DEMOA\_H#define DEMOA\_Htemplate<class T>class DemoA{public:    DemoA(){}               //假设T类型可进行默认初始化（成员变量t会进行默认初始化）    DemoA(const T &val):t(val){}    //假设T类型可进行拷贝初始化    bool operator<(const T &ele){        return this->t < ele.t;       //假设T类型支持<运算符    }private:    T  t;};#endif |

|  |
| --- |
| //A.h#ifndef A\_H#define A\_H//类A仅支持默认初始化、拷贝初始化，不支持<运算符class A{public:    A(){}    A(const A &a){}};#endif |

|  |
| --- |
| //test1.cpp#include "A.h"#include "DemoA.h"void test1(void){    DemoA<A>  obj;    //正确，因为只使用到了DemoA<T>的默认构造函数}//编译器在当前源文件中实例化出DemoA<A>的等价类（该类可作为类模板Demo的一个实例化版本）template< >class  DemoA<A>{public:    DemoA(){}private:    A  t;}; |

|  |
| --- |
| //test2.cpp#include "A.h"#include "DemoA.h"void  test2(void){    DemoA<A>  obj1;   //使用到了DemoA<T>的默认构造函数    DemoA<A>  obj2;   //使用到了DemoA<T>的默认构造函数    if(obj1 < obj2){     //使用到了DemoA<T>的<运算符（编译报错）    }}//编译器在当前源文件中实例化出DemoA<A>的等价类（该类可作为类模板Demo的一个实例化版本）template< >class  DemoA<A>{public:    DemoA(){}    bool operator<(const A &ele){        return this->t < ele.t;       //编译报错，因为类A不支持<运算符    }private:    A  t;}; |

## 3.2 模板代码的组织方式

        在编译阶段，当我们调用一个普通函数或创建并使用一个类对象时，编译器只需要掌握该函数的声明或该类的定义，而普通函数的定义或类的成员函数的定义则不必可见（在链接阶段，编译器会根据普通函数或类的名称去搜索相应的普通函数定义或类的成员函数定义）；

        但是模板则不同：为了在编译阶段生成一个模板的实例化版本，编译器需掌握函数模板或类模板成员函数的定义，因此，模板的头文件通常既包括声明也包括定义（即函数模板和类模板成员函数的定义通常都放在头文件中）。

# 4 函数模板

## 4.1 函数模板实参类型自动推断

        函数实参推断基本思想：用已知的事情推断未知的事情。

        因为对于函数模板的模板形式参数在实践中都会应用于函数参数列表中作为函数形参类型，所以编译器可以利用调用中的函数实参来确定其模板实参。从模板函数调用的实参来确定函数模板的实参的过程被称为模板实参推断（template argument deduction）。在模板实参推断过程中，编译器使用函数调用中的实参类型来寻找模板实参，使得用这些模板实参生成的模板函数版本与给定的函数调用最为匹配。

|  |
| --- |
| //函数模板声明template<class T>bool compare(const T &, const T &);void test(){    int a = 23;    int b = 32;    //等价于调用compare<int>(a, b);    compare(a, b);} |

### 4.1.1 函数实参与类型为模板类型参数的函数形参之间的类型转换

        和非模板函数一样，我们在一次不显式指定函数模板实参的调用中传递给模板函数的实参被用来初始化该模板函数的形参，但是模板函数的形参初始化规则和非模板函数的形参初始化规则并不相同：

        （1）非模板函数的形参在被不同类型的实参初始化时，遵循通用的隐式类型转换规则；

        （2）而模板函数的形参在被不同类型的实参初始化时，只有以下三种与类型说明符无关的隐式类型转换能自动地应用于这些实参：

|  |  |
| --- | --- |
| 类型转换 | 说明 |
| 顶层const（Top-Level const）被忽略 | 在函数参数传递的过程中，无论是函数形参还是函数实参的顶层const都会被忽略；（注：当函数形参为非引用类型（如T）时，才会忽略函数实参的顶层const） |
| 底层const（Low-Level const）转换 | 可以将一个非const对象的引用（或指针）的模板函数实参传递给一个const的引用（或指针）的模板函数形参； |
| 数组或函数指针转换 | 如果模板函数形参是非引用类型，则可以对数组或函数类型的实参应用正常的指针转换。注1：若为引用类型，则形参可以直接接收数组或函数对象；注2：一个数组实参可以隐式转换为一个指向其首元素的指针；而一个函数实参可以转换为一个该函数类型的指针。 |

        在完成了上述有限的隐式类型转换后，编译器才会根据转换后的结果推断出函数模板的实参，最后生成一个新的模板实例，也就是说，编译器根据模板函数的实参类型得到模板函数应该使用的形参类型，从而推断出与该形参类型相关的模板类型参数的实参值，如下所示：

|  |
| --- |
| //函数模板声明template<class  T>T  fobj(T, T);template<class  T>T  fref(const  T&, const  T&);void test(){    string s1("abc");    const string s2("cba"); //顶层const    /\*  s1为string类型，而模板函数形参类型为T（而不是const T&），所以编译器无法对其进行隐式类型转换；        s2为顶层const，编译器忽略其顶层const属性后，将s2看成是string类型，即s1和s2都是string类型，        根据函数实参类型string（实参经隐式类型转换后的实际类型）和函数形参类型T的对应关系，编译器可推断出模板形参T的实参为string，        因此最终调用的是fobj(string, string)函数\*/    fobj(s1, s2);       // string  fobj<string>(string, string)    /\*  s1为string类型，而模板函数形参类型为const T&，所以编译器可以将s1从string隐式转换为const string &类型；        s2为顶层const，因为形参类型为引用，所以编译器不会忽略其顶层const属性，之后就和参数s1的情况一样，        因为模板函数形参类型为const T&，所以编译器可以将s2从const string隐式转换为const string &类型，        根据函数实参类型const string &（实参经隐式类型转换后的实际类型）和函数形参类型const T &的对应关系，        编译器可推断出模板形参T的实参为string，因此最终调用的是fref(const string &, const string &)函数\*/    fref(s1, s2);       // string  fref<string>(const string &, const string &)    int a[10], b[32];    /\*  函数实参a（或b）是int数组，而模板函数形参类型为T（而不是const T&），        所以编译器可以将实参a（或b）隐式转换为指向其首元素的指针类型int \*，        所以最终编译器可由函数实参类型int \*与函数形参类型T的对应关系，        推断出模板参数T的实参为int \*类型\*/    fobj(a, b);     // int \*fobj<int \*>(int \*, int \*)    /\*  函数实参a是数组类型int[10]，而模板函数形参类型为const T&引用，所以编译器可以将实参a隐式转换为const int[10]，        所以最终编译器可由函数实参类型const int[10]与函数形参类型const T &的对应关系，推断出模板参数T的实参为int [10]类型；        同理可根据函数实参b推断出模板参数T的实参为int [32]类型，二者前后矛盾，故编译报错\*/    fref(a, b);     //编译报错} |

        注1：因为对于模板函数的调用，编译器可进行的实参的隐式类型转换实在是太有少了，所以在大部分的场景中，编译器由于根本没有机会进行隐式类型转，而直接推断函数模板实参类型后生成模板实例。

        注2：因为一个模板类型参数可以用作多个函数形参的类型，而又因为在模板机制下，并不会进行与类型说明符相关的隐式类型转换，所以若模板函数的多个形参都使用了同一个模板类型参数作为类型，那么传递给这些函数形参的实参（在进行隐式类型转换后）必须具有相同的类型，否则，由不同函数实参（与这些实参相对应的形参使用了同一个模板类型参数作为类型）推断出的模板类型参数的实参就不相同，使得编译报错，如下所示：

|  |
| --- |
| //函数模板声明template<class T>bool compare(const T &, const T &);     //模板函数的两个形参类型相同void test(){    int a = 23;    long b = 32;    int c = 17;    /\*  先分别将函数实参a和b隐式类型转换为const int和const long类型，        然后分别由函数实参a和b的类型与其形参类型的对应关系，        推断出模板类型参数T的类型分别为int和long，        即二者推断出的模板参数T的类型不相同，因此编译报错\*/    compare(a, b);      //编译报错    //函数实参a和c的类型相同，都为int，最终推断出模板类型参数T的类型为int    compare(a, c);} |

如果希望允许模板函数的实参使用不同类型说明符的类型，我们可以分别为不同的函数形参使用不同的模板类型参数作为类型，如下所示：

|  |
| --- |
| //函数模板声明template<class T1, class T2>bool compare(const T1 &, const T2 &);       //模板函数的两个形参类型相同void test(){    int a = 23;    long b = 32;    //由函数实参a推断出模板类型参数T1的类型为int，由函数实参b推断出模板类型参数T2的类型为long    compare(a, b);} |

注3：在函数模板中可混合使用模板类型参数和普通类型作为函数参数的类型，此时对于那些使用普通类型的函数形参，在其实参传递过程中，则可以进行正常的隐式类型转换规则进行实参到形参的隐式类型转换。同样地，对于显式指定模板类型参数实参的模板函数调用，也可进行正常的隐式类型转换。如下所示：

|  |
| --- |
| //函数模板声明template<class T>bool compare(const T &, const int &);   //第一个形参使用模板类型参数，第二个形参使用普通类型intvoid test(){    int a = 23;    long b = 32;    //因为第二个形参类型为普通类型int，所以函数实参b可从long隐式转换为const int类型    compare(a, b);} |

|  |
| --- |
| //函数模板声明template<class T>bool compare(const T &, const T &);     //模板函数的两个形参类型相同void test(){    int a = 23;    long b = 32;    /\*  显式指定模板类型参数T的实参值为int，因此函数声明变为"bool compare(const int &, const int &);"，        所以函数实参b可从long隐式转换为const int类型\*/    compare<int>(a, b);} |

         注4：对于“由编译器隐式推断模板函数类型参数实参值”和“由用户显式指定模板函数类型参数实参值”的过程，二者的编译流程是不相同的，从而导致部分特性的不一样（如只有对于显式指定模板类型参数实参的模板函数调用，才可进行正常的隐式类型转换），如下所示：

        （1）由编译器隐式推断模板函数类型参数实参值的编译流程：调用模板函数-->由模板函数调用实参自动推断出模板函数模板实参值-->判断推断出的模板实参值是否可用-->使用推断出的模板实参值，生成模板函数定义代码。

        （2）由用户显式指定模板函数类型参数实参的编译流程：调用模板函数-->根据显式给定的模板函数实参生成模板函数定义代码-->判断调用模板函数时传递的实参是否正确。

### 4.1.2 函数形参类型为模板形参的引用类型时的模板实参推断

        当函数模板的函数形式参数类型为引用（包括非const左值引用（T &）、const左值引用（const T &）以及右值引用（T &&））时，会按照以下步骤对模板形参T的实参值进行推断：

        （1）因为函数模板形参类型为引用，所以只有函数形参的顶层const可以被忽略；

        （2.1）编译器会先使用引用绑定的通用规则来判断在函数模板调用时传递的实参是否可以被形参引用所绑定，对于不满足正常引用绑定规则的函数实参，编译器无法生成模板函数实例，故编译报错；

        （2.2）若函数模板的形参类型为右值引用（T &&），编译器除了使用引用绑定的通用规则来判断在函数模板调用时传递的实参是否可以被形参引用所绑定外，还使用了一个引用绑定的特例规则（即“**右值引用可以绑定到（const/非const）左值对象**”）对此进行判断；

        （3.1）对于通过引用绑定的通用规则判断的函数实参，编译器就会根据函数模板形参类型和函数实参类型的对应关系直接推断出模板类型参数的实参值；

        （3.2）对于通过引用绑定的特例规则判断的函数实参（注：此时的函数形参类型肯定为右值引用T &&，函数实参类型肯定为（const/非const）左值对象），**编译器直接将模板类型参数T的实参值推断为与函数实参类型对应的左值引用类型**，然后根据引用折叠规则来确定需要实例化出来的模板函数的形参类型；

注“：根据上述（2.1）和（2.2）可知，若函数模板的形参类型为右值引用（T &&），既可以使用正常的引用绑定规则（“右值引用只能绑定到右值对象”），又可以使用例外的引用绑定规则（“右值引用可以绑定到（const/非const）左值对象”），概括起来即为“**右值引用可以绑定到任意的对象（const/非const的左/右值对象）**”

4.1.2.1 模板实参推断规则汇总表

        如下所示为对上述过程总结的示例表格：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 函数形参类型 | 函数实参类型 | 推断出T的实参值 | 函数形参实际类型 | 备注 |
| T & | int（左值） | int | int &（左值） | 函数实参必须为左值 |
| const int（左值） | const int | const int &（左值） |
| const T & | int（左值/右值） | int | const int &（左值） | 函数实参可以为任意类型（左值/右值） |
| const int（左值） | int | const int &（左值） |
| T && | int（右值） | int | int &&（右值） | 函数实参可以为任意类型（左值/右值） |
| int（左值） | int & | int & && -->（引用折叠） int &（左值） |
| const int（左值） | const int & | const int & && -->（引用折叠） const int &（左值） |

4.1.2.2 引用绑定的通用规则

（1）非const左值引用只能绑定到非const的左值对象

（2）const左值引用可以绑定到任意的对象（const/非const的左/右值对象）

（3）右值引用只能绑定到右值对象（临时对象或字面值常量对象）

4.1.2.3 引用折叠规则

如果我们间接地创建了一个引用的引用，那么这些引用将发生“折叠”作用，如下所示：

（1）类型X& &（左值引用的左值引用）、X& &&（左值引用的右值引用）和X&& &（右值引用的左值引用）最终都会被折叠成X &；

（2）类型X&& &&（右值引用的右值引用）最终会被折叠成X &&；

注：引用折叠只能应用于间接创建的引用的引用，如类型别名和模板参数。

4.1.2.4 函数形参类型为模板形参的左值引用（T &或const T &）

|  |
| --- |
| template<typename T>void  demo(T &param){}      //函数模板的形参类型T&形式为非cosnt左值引用void  test(){    const int c = 32;       //c为const左值对象    /\*  模板函数调用实参为const左值对象，而根据正常引用规则“非const左值引用只能绑定到非const左值对象”，        所以当非const左值引用形参param绑定到const实参c时，模板类型参数T的实参肯定包含const，        故最后实例化出模板函数：void demo<const int>(const int &);\*/    demo(c);} |

4.1.2.5 函数形参类型为模板形参的右值引用（T &&）

|  |
| --- |
| //C++primer上的实现template <class T>typename remove\_reference<T>::type&&  move(T&& arg) noexcept{    //假设arg的基本类型为type，则T的三种可能实参值：type、type &和const type &    return  static\_cast<remove\_reference<T>::type&&>(arg);}/\*    函数实参a为左值（int），直接将T推断为"int &"，    所以函数形参arg的类型为"int& &&"-->引用折叠-->"int &"    实例化版本：    typename remove\_reference<int &>::type&&  move(int &arg) noexcept;\*/int a = 32;std:move(a);/\*    函数实参b为const左值（const int），直接将T推断为"const int &"，    所以函数形参arg的类型为"const int& &&"-->引用折叠-->"const int &"    实例化版本：    typename remove\_reference<const int &>::type&&  move(const int &arg) noexcept;\*/const int b = 32;std::move(b);/\*    函数实参32为右值（int），直接比较"int&&"和"T&&"，推断出T为"int"，    所以函数形参arg的类型为"int &&"    实例化版本：    typename remove\_reference<int>::type&&  move(int &&arg) noexcept;\*/std::move(32); |

### 4.1.3 函数形参类型为模板形参的非引用类型时的模板实参推断

        当函数模板的形参类型为非引用类型时（如T），则实例化出的模板函数的形参为使用值传递的类型参数（而非引用参数），所以直接对模板函数的实参和函数模板的形参进行比较，从而推断出模板参数的实参值即可。

## 4.2 显式指定函数模板实参

在如下的情况下，编译器无法根据函数调用的实参自动推断出函数模板实参的类型：（1）函数模板的返回类型和模板类型参数相关联，但该返回类型和函数模板的函数形参类型没有任何关系；（2）函数模板的函数形参类型均与模板参数无关；（3）函数模板的函数形参类型是经模板类型参数间接转换得到（而不直接与模板类型参数相关联），如：

|  |
| --- |
| template <class T>T&& forward (typename remove\_reference<T>::type&& arg) noexcept; |

### 4.2.1 显式模板实参匹配规则

        显式模板实参在尖括号中给出，位于函数名之后，实参列表之前，显式模板实参按由左到右的顺序与对应的模板形参匹配：第一个模板实参与第一个模板形参匹配，第二个模板实参与第二个模板形参匹配，依次类推。且在尖括号中，只有尾部（最右）模板形参的模板实参可以被省略，但是前提是它们的实参值可以由编译器从模板函数的实际参数推断出来。如下所示：

|  |
| --- |
| template<class  T1, class  T2>T1  func(T2  param){    return param;}void test(){    //混合使用显式模板实参和隐式模板实参    //显式指定模板类型参数T1的实参值为int，而模板类型参数T2的实参值可由函数实参32推断得到为int    auto retn = func<int>(32);} |

### 4.2.2 显式模板实参应用场景

|  |
| --- |
| class Sim3Transform{    ...    //Sim3Transform to g2o    //note: you must specify g2o::Sim3 explicitly as argument of template parameter G2O\_SIM3    template<typename G2O\_SIM3>    G2O\_SIM3 toG2o() const    {        H\_PRECISION s = getScale();        Eigen::Matrix3d R = getRotation();        Eigen::Vector3d T = getTranslation();        return G2O\_SIM3(R, T, s);    }    ...};Sim3Transform sim3;g2o::Sim3 g2oSim3 = sim3.toG2o<g2o::Sim3>(); |

# 5 类模板

        与函数模板的不同之处是，编译器不能为类模板推断模板实参类型，因此当我们使用模板类类型时，必须显式指定模板实参（但这一规则有一个例外：在类模板自己的作用域中，我们可以直接使用该模板名而不提供模板实参）。

# 5.1 类模板的成员

        一方面，因为类模板的每个实例都有其自己版本的成员（包括非静态成员函数、静态成员函数和静态成员变量（注：因为非静态成员变量不能在类模板外部进行定义，所以在此不予以讨论）），所以类模板的成员具有和模板相同的模板参数（即使类模板成员没有用到任何模板参数，如析构函数），因此，定义在类模板外部的成员必须以关键字template开始，后接类模板参数列表（以此来声明模板形参）；——说明定义在类模板外部的成员都需要添加"template<模板形参列表>"前缀

        另一方面，因为我们在类的外部定义其成员时，都必须通过“类名”+作用域运算符（::）来说明该成员是属于哪个类的，而对于类模板来说，这个“类名”就是模板类的名称，我们通过类模板及其模板实参获取模板类的名称，不过需要注意的是，此时模板实参与模板形参是相同的；——说明定义在类模板外部的成员都需要通过"类模板名称<模板实参列表>::"指定后续的定义语句将位于指定模板类作用域内

### 5.1.1 类模板的非static成员函数

        与其他任何非模板类相同，我们既可以在类模板的内部，也可以在类模板外部为其定义成员函数（注：即使在类模板的外部定义成员函数，但根据良好的编码规范要求，类模板的声明和定义都应该放在同一个文件（同一个头文件或源文件）中），且定义在类模板内的成员函数仍然是被隐式声明为内联函数；

        综上所述，当我们在模板类外部定义其成员函数时，有如下所示的代码：

|  |
| --- |
| //Demo.htemplate<typename T>class  Demo{public:    Demo();    ~Demo();    bool  compare(const T &obj) const;    void  print(const string &msg);private:    T  ele;};//即使在成员函数中没有使用到任何模板参数，仍需要提供模板形参列表template<typename T>      // "template<typename T>"中的T为模板形参Demo<T>::Demo(){}     // "Demo<T>"中的T为模板实参template<typename T>Demo<T>::~Demo(){}template<typename T>bool  Demo<T>::compare(const T &obj) const{    return this->ele < obj.ele;}template<typename T>void  Demo<T>::print(const string &msg){    cout << msg << endl;} |

### 5.1.2 类模板的static成员

        和非模板类一样，在类模板中也可以定义static成员，非模板类中的static成员从属于其所属的非模板类类型，而类模板中定义的static成员则从属于其实例化后的模板类类型（注：类模板不是类型，而模板类是类型），所以当我们通过模板类名引用其static成员时，必须显式指定相应的模板实参。

        需要注意的是，和非模板类的静态数据成员需要定义在源文件中不同，如前文所述的由于需要保证模板在实例化时其定义是可见的，所以类模板的静态数据成员的定义仍需和其声明绑定在一起放在同一个文件（头文件或源文件）中，如下所示：

|  |
| --- |
| // Demo.htemplate<typename T>class Demo{public:    static T count();private:    static T count\_;};template<typename T>T Demo<T>::count(){    return count\_;}//注：类模板的静态数据成员的定义放在头文件也不会导致重定义问题template<typename T>T  Demo<T>::count\_ = 32;      //模板类静态数据成员定义及初始化仍需放在同一个文件中 |

## 5.2 在类模板中使用自身模板的名字

        当我们使用一个模板类类型时必须提供模板实参，但这一规则有一个例外：若在类模板自己的作用域中，我们可以直接使用类模板名而不用提供模板实参，此时编译器将假定我们使用的模板类类型的模板实参与该模板类在实例化时使用的模板实参完全一致（暗示着如果我们所需的模板实参与该模板类在实例化时使用的模板实参不一致，那我们仍需要显式指定该模板类类型的模板实参）。

        注：上述的“可以”一词意味着即使繁琐地提供显式的模板实参也是正确的。

### 5.2.1 在类模板的内部使用自身类模板名字

        若在类模板的内部，相当于我们已处于类模板自己的作用域中，在这种情况下，编译器在处理模板自身名字时就好像我们已经提供了与模板参数匹配的实参一样，如下所示：

|  |
| --- |
| // Demo.htemplate<typename T>class  Demo{public:    //在类模板Demo<T>内部可直接使用自身模板类类型名称Demo，而不需要写出Demo<T>    //注：即使函形参obj的类型写成"const Demo<T> &"也是正确的    bool  operator<(const Demo &obj){        return  this->ele < obj.ele;    }    //前缀++运算符    Demo  &operator++(){        ++this->ele;        return  \*this;    }private:    T  ele;}; |

### 5.2.2 在类模板的外部使用自身类模板名字

        当我们在类模板外定义其成员时，必须记住，我们并不在类的作用域中，直到遇到类名+作用域运算符才表示进入类的作用域，也就是说，在类的作用域外，仍必须为自身类模板名显式指定模板实参，如下所示：

|  |
| --- |
| // Demo.htemplate<typename T>class  Demo{public:    //在类模板Demo<T>内部可直接使用自身模板类类型名称Demo，而不需要写出Demo<T>    //注：即使函形参obj的类型写成"const Demo<T> &"也是正确的    bool  operator<(const Demo &obj);    //前缀++运算符    Demo  &operator++();private:    T  ele;};//定义形参obj时已处于由"Demo<T>::"限定的模板类作用域中，因此可以省略自身模板类类型Demo的模板实参template<typename  T>bool  Demo<T>::operator<(const Demo &obj){    return  this->ele < obj.ele;}//在函数返回类型位置还未处于由"Demo<T>::"限定的模板类作用域中，因此属于模板类作用域外部，所以需要显式指定模板实参Ttemplate<typename  T>Demo<T>  &Demo<T>::operator++(){    ++this->ele;    return  \*this;}//通过函数尾置返回类型可以使函数返回类型处于由"Demo<T>::"限定的模板类作用域中，因此可以省略自身模板类类型Demo的模板实参template<typename  T>auto  Demo<T>::operator++()->Demo &{    ++this->ele;    return  \*this;} |

## 5.3 类模板别名

### 5.3.1 用typedef定义实例化后的模板类的类型别名——本质为模板类

        由类模板实例化出的模板类定义了一个类类型，与任何其他类类型一样，我们可以定义一个typedef来引用模板类类型，如下所示：

|  |
| --- |
| typedef vector<int> iVector; |

        需要注意的是，因为模板不是一个类型，所以我们不能定义一个typedef去引用一个模板，即我们不能定义一个typedef去引用vector或vector<T>，如下所示：

|  |
| --- |
| typedef vector vec; //编译报错：invalid use of template-name ‘std::vector’ without an argument listtemplate<class T>typedef vector<T> vec<T>;   //编译报错：template declaration of ‘typedef’//编译报错：expected unqualified-id before ‘template’typedef template<class T> vector<T> vec<T>; |

### 5.3.2 用using定义类模板的别名（C++11）——本质为类模板

        虽然我们不能用typedef为类模板定义模板别名，但C++11标准允许我们使用using为类模板定义一个模板别名，如下所示：

|  |
| --- |
| template<typename T>using Twin = pair<T, T>;      //Twin为类模板void  test(){    Twin<string>  author; //author是一个pair<string, string>} |

        另外，当我们定义一个类模板别名时，可以固定一个或多个模板参数，如下所示：

|  |
| --- |
| template<typename T>using partNo = pair<T, int>;  //固定pair的第二个模板实参值为intvoid test(){    //使用类模板partNo时，无法指定pair第二个模板参数的实参值，否则编译报错：wrong number of template arguments (2, should be 1)    partNo<string>  books;        //books是一个pair<string, int>} |

|  |
| --- |
| namespace Eigen{//使用Eigen名称空间下的vector即可保证内存对齐template<class T, class Alloc = Eigen::aligned\_allocator<T>>using vector = std::vector<T, Alloc>;}void test(){    //因为使用Eigen名称空间下的vector即可保证内存对齐，所以无须为Eigen::vector显式指定Alloc模板实参    Eigen::vector< Eigen::Vector3d>  vPoint;    ...} |

## 5.4 成员模板

一个类（无论是普通类还是类模板）可以包含函数模板作为其成员函数，这种成员被称为成员模板

### 5.4.1 成员模板不能是虚函数

        因为编译器期望在处理类的定义的时候就能确定这个类的虚函数表的大小，如果允许有类的虚成员模板函数，那么就必须要求编译器提前知道程序中对该类的该虚成员模板函数的所有调用，而这是不可行的（存在实现上的困难和复杂性），所以C++标准不允许成员模板是virtual的。

### 5.4.2 类模板的成员模板

        对于类模板，我们也可以为其定义成员模板，在这种情况下，类模板和成员模板各自有自己的、相互独立的模板参数；另外，当我们在类模板的外部定义一个成员模板时，必须同时为类模板和成员模板提供模板参数列表，且类模板的模板参数列表在前，成员模板的模板参数列表在后，如下所示：

|  |
| --- |
| // Blob.htemplate<typename T>class Blob{public:    template<typename It>    Blob(It begin, It end); //构造函数为成员模板    ...};template<typename T>  //类模板模板参数列表template<typename It> //成员模板模板参数列表Blob<T>::Blob(It begin, It end){    ...} |

5.4.2.1 类模板的成员模板的实例化

        为了实例化一个类模板的成员模板，我们必须同时提供类和函数模板的模板实参，而在实践中，成员模板的模板实参通常都是由编译器自动推断，如下所示：

|  |
| --- |
| template<typename T>class Blob{public:    //构造函数为成员模板    template<typename It>    Blob(It begin, It end);    //成员函数为成员模板    template<typename I>    void func(I a);    //静态成员模板    template<typename I>    static void sfunc(I a);};template<typename T>  //类模板模板参数列表template<typename It> //成员模板模板参数列表Blob<T>::Blob(It begin, It end){}template<typename T>  //类模板模板参数列表template<typename I>  //成员模板模板参数列表void Blob<T>::func(I a){}template<typename T>  //类模板模板参数列表template<typename I>  //成员模板模板参数列表void Blob<T>::sfunc(I a){}int main(){    vector<int> v;    //只能由编译器自动推断构造函数模板实参    Blob<int> b(v.begin(), v.end());    b.func(v);  //等价于：b.func<decltype(v)>(v);    Blob<int>::sfunc<char>(32);    return 0;} |

# 6 模板特例化（template specialization）

        模板的一个特例化版本就是使模板中的一个或多个模板参数被指定为特定的类型（对应模板类型参数）或值（对应模板值参数）时的模板的一个独立的定义。

注：在定义原模板的一个实例化版本前，原模板的声明必须可见。

## 6.1 函数模板特例化

        当我们特例化一个函数模板时，必须为原函数模板中的所有模板形式参数都提供实参。为了指出我们正在实例化一个模板，应使用关键字template且紧跟着一个空尖括号对（<>），其中空尖括号对用来指出我们将显式为原函数模板中的所有模板形参提供实参，如下所示：

|  |
| --- |
| template<typename T>int compare(const T &v1, const T &v2){    if (v1 < v2) return -1;    if (v2 < v1) return 1;    return 0;}/\*  分析：    （1）我们希望当调用函数模板compare的函数实参类型为const char \*时，编译器使用我们定义的相应的特例化版本进行实例化，因此，需要先确定当使用类型为const char\*的函数实参调用函数模板compare时，经编译器模板实参自动推断后，实例化出的模板函数原型以及对应的模板实际参数值是什么，然后再定义为原函数模板指定该模板实参值后的特例化版本（使用相同的模板函数原型即定义了指定模板实参值的特例化版本）；    （2）当函数实参类型为"const char\*"时，模板类型参数T的实参值为"const char \*"，表示一个指向char类型常量的指针类型，也就是说此时"const T &"表示一个指针类型常量的引用，而该指针类型指向char类型常量，因此实例化出的模板函数形参类型为"const char \*const &"，表示一个指向char类型常量的指针常量的引用（模板类型参数和指针、const混合使用）；    （3）所以我们需要为原函数模板定义一个模板函数参数类型为"const char \*const &"的特例化版本，以告诉编译器当模板类型参数T的实参值为"const char\*"时，使用该特例化版本进行实例化\*/template< >int compare(const char \* const &p1, const char \* const &p2){    return std::strcmp(p1, p2);}//上下两种书写形式相互等价template< >int compare<const char\*>(const char \* const &p1, const char \* const &p2){    return std::strcmp(p1, p2);}int main(){    const char \* p1 = "abc";    const char \* p2 = "cba";    compare(p1, p2);        //输出：compare<const char\* const>()    //结论：字面值常量"abc"并不是const char \*类型，而是"const char[4]"数组类型    compare("abc", "cba");  //输出：default compare()    return 0;} |

## 6.2 类模板特例化

        除了特例化函数模板，我们还可以特例化类模板，和函数模板的特例化不同，类模板既可以进行全特例化（和特例化函数模板时一样，在特例化时为原模板中所有模板形参指定实参值，从而生成一个特定的模板类），也可以进行部分特例化（即在特例化时只为原模板中部分模板形参指定实参值（生成一个新的类模板），或只为类模板中的部分成员指定模板实参值）。

### 6.2.1 类模板的全特例化

6.2.1.1 定义hash<T>的全特例化版本

        类模板的全特例化可用于对标准库中的模板类根据实际需求，特例化出使用指定模板实参的模板类：如在默认情况下，无序关联容器使用以其关键字类型key\_type为模板实参的hash模板类hash<key\_type>来生成元素的哈希值（通过为无序关联容器的模板形式参数Hash指定默认实参值hash<Key>来实现）。但是对于绝大部分的类类型（特别是由用户自定义的类类型），都无法生成相应的hash<key\_type>模板类（因为类模板hash<T>不存在任何定义语句，所有可实例化的hash模板类都必须通过定义相应的特例化版本来实现）。因此，若我们希望以自定义的类类型作为键去使用无序关联容器（而又不希望通过定义并给无序关联容器传递该类型的哈希可调用对象的方式来实现），那么我们需要定义以该自定义类类型为模板实参的hash<T>的特例化版本。

        需要注意的是，由标准库定义的所有hash<T>的特例化版本，都满足以下三个条件：

        （1）包含一个接受一个参数，并返回size\_t类型的重载的调用运算符（用于计算并返回指定类型输入对象的hash值）；

        （2）包含两个类型成员，result\_type（hash值类型）和argument\_type（需要计算hash值的对象类型），分别对应于上述调用运算符的返回类型和参数类型，且argument\_type在默认情况下应该定义了==重载运算符；

        （3）默认构造函数和拷贝赋值运算符（可使用隐式定义）。

        为了最大限度地降低风险，我们为自定义类类型定义的hash<T>的特例化版本也应该满足上述三个基本条件（因为上述的三个基本条件已被标准库的其他组件（包括无序关联容器）作为使用hash<T>的编程假设，所以若我们定义的hash<T>的特例化版本不满足上述的三个条件，可能因为不满足无序关联容器对hash<T>的编程假设，而在实例化无序关联容器时编译报错）。

        另外，因为std::hash<T>类模板定义在名称空间std中，所以我们定义的特例化版本也必须在std名称空间中（特例化版本与原模板必须位于同一个名称空间中），其如下所示：

|  |
| --- |
| #ifndef STUDENT\_H#define STUDENT\_H#include <functional>class Student{    //定义模板类std::hash<Student>为Student的友元类    friend struct std::hash<Student>;public:    bool operator ==(const Student &s) const    {        return this->name == s.name &&               this->age == s.age &&               this->height == s.height;    }private:    std::string name;    int age;    double height;};//为了让Student的用户能使用hash的特例化版本，我们应该在Student的头文件中定义该特例化版本namespace std {//在std名称空间中定义hash<T>的一个全特例化版本template<>struct hash<Student>{    typedef size\_t result\_type;    typedef Student argument\_type;  //在默认情况下，此类型需定义==重载运算符    //在类的内部进行定义    size\_t operator()(const Student &s) const    {        //因为模板类hash<Student>为Student的友元，所以在hash<Student>的内部可以直接访问Student对象的私有成员        //使用异或运算组合对象的多个数据成员的hash值作为对象的hash值        return hash<std::string>()(s.name) ^               hash<int>()(s.age) ^               hash<double>()(s.height);    }};}   //名称空间的右花括号之后不需要分号#endif |

### 6.2.2 类模板的部分特例化

6.2.2.1 部分特例化类模板

        当我们需要为一个类模板定义部分特例化版本时，需使用以下方式进行定义：

        （1）由于类模板的一个部分特例化版本本质仍是一个类模板，所以在定义时仍需要定义模板参数列表（如"template<typename T>"），且对于原模板中每个未完全确定类型的模板参数，在部分特例化版本的模板参数列表中都有一项与之对应——即部分特例化版本的模板参数列表是原始模板参数列表的一个子集或者是一个特例化版本；

        （2）类似于任何其他特例化版本，部分特例化版本的名字仍与原模板的名字相同；

        （3）类似于任何其他特例化版本，在类名之后，我们要在尖括号对中为特例化的模板形式参数指定实际参数，且这些实参与原始模板模板参数列表（而不是所在的部分特例化版本的参数列表）中的参数位置向对应（也就是说原始模板模板参数列表中有几个参数，在部分特例化版本中就要指定几个模板实参值），如下所示：

        注：因为类模板的部分特例化版本仍然是一个类模板，所以和任何其他类模板一样，类模板的部分特例化版本也应该定义在头文件中（而不是定义在源文件）。

|  |
| --- |
| //Demo.h#ifndef Demo\_H#define Demo\_H#include "iostream"using namespace std;//原始模板声明（注：模板特例化机制下，原始模板的定义不是必须的）template<typename T1, typename T2>class Demo;//我们不能定义原模板的第一个模板参数和第二个模板参数都为任意类型时的部分特例化版本，因为这个特例化版本相当于要覆盖原模板的作用，所以编译器不允许这种行为//编译报错：partial specialization ‘Demo<T1, T2>’ does not specialize any template argumentstemplate<typename T1, typename T2>class Demo<T1, T2>{public:    void run(){        cout << "Demo<T1, T2>" << endl;    }};//当原模板的第一个模板参数T1的实参为int，第二个模板参数T2的实参为任意类型时的部分特例化版本template<typename T>class Demo<int, T>{public:    void run(){        cout << "Demo<int, T>" << endl;    }};//当原模板的第一个模板参数T1的实参为任意类型，第二个模板参数T2的实参为int时的部分特例化版本template<typename T>class Demo<T, int>{public:    void run(){        cout << "Demo<T, int>" << endl;    }};//当原模板的第一个模板参数T1的实参为任意类型的左值引用，第二个模板参数T2也为任意类型的左值引用时的部分特例化版本template<typename T1, typename T2>class Demo<T1&, T2&>{public:    void run(){        cout << "Demo<T1&, T2&>" << endl;    }};//当原模板的第一个模板参数T1和第二个模板参数T2的实参为相同的任意类型的左值引用时的部分特例化版本template<typename T>class Demo<T&, T&>{public:    void run(){        cout << "Demo<T&, T&>" << endl;    }};//当原模板的第一个模板参数T1的实参为任意类型的数组，第二个模板参数T2的实参为int时的部分特例化版本template<typename T>class Demo<T[], int>{public:    void run(){        cout << "Demo<T[], int>" << endl;    }};#endif |

|  |
| --- |
| //main.cpp#include "Demo.h"int main(){    Demo<int, double> d1;    d1.run();   //输出：Demo<int, T>    Demo<double, int> d2;    d2.run();   //输出：Demo<T, int>    Demo<int &, double &> d3;    d3.run();   //输出：Demo<T1&, T2&>    Demo<int &, int &> d4;    d4.run();   //输出：Demo<T&, T&>    Demo<int[], int> d5;    d5.run();   //输出：Demo<T[], int>    return 0;} |

6.2.2.2 特例化类模板中的成员

        我们也可以只特例化类模板中特定成员函数而不是特例化整个模板，例如，假设Foo是一个类模板，包含一个成员函数test()，那么我们可以只特例化该成员，需要注意的是，当我们只是特例化类模板中的某个成员函数时，只能在类外部定义该成员函数的特例化版本，而因为该特例化版本相当于一个普通非模板函数，因此为了避免编译重定义错误，必须将该成员函数的特例化版本定义在源文件中（而不是头文件），如下所示：

|  |
| --- |
| //Foo.h#ifndef FOO\_H#define FOO\_H#include "iostream"using namespace std;template<typename T>struct Foo{    void test(){        cout << "default test()" << endl;    }};#endif |

|  |
| --- |
| //foo.cpp#include "Foo.h"//成员函数的特例化版本只能定义在源文件中（避免编译重定义错误）template<>void Foo<int>::test(){    cout << "Foo<int> test()" << endl;} |

|  |
| --- |
| //main.cpp#include "B.h"int main(){    Foo<int> f1;    f1.test();    //调用特例化版本的test()，输出：Foo<int> test()    Foo<long> f2;    f2.test();    //调用通用版本的test()，输出：default test()    return 0;} |

## 6.3 模板特例化允许原模板没有定义

        因为模板的特例化版本是用于在指定模板实参的条件下实例化时使用的代码，所以模板的实例化版本的定义与原模板的定义相互独立，也就是说模板特例化版本中的定义语句可与原模板中的定义语句没有任何联系，因此，在定义模板的一个特例化版本时，并不需要知道原模板的定义（但为了知道模板形参的类型与数目，必须知道原模板的声明）。

        综上所述，C++标准允许对一个不存在任何定义的模板进行特例化，这样带来的额外作用是：原模板只允许那些存在相应特例化版本的模板实参进行实例化（注：对于包含定义的模板，只要用户传递的模板实参满足该模板对模板实参所需的编程假设，编译器都可以使用该模板实参对该模板进行实例化），否则编译报错，如下所示：

|  |
| --- |
| #include <functional>using namespace std;//注：标准库中的哈希类模板hash<T>不存在任何定义语句，所有可实例化的模板类都必须通过定义其实例化版本来实现int main(void){    hash<vector<int>>  h1;  //编译报错：aggregate ‘std::hash<std::vector<int> > h1’ has incomplete type and cannot be defined    hash<int>  h2;            //编译成功，因为标准库已为hash<int>定义了特例化版本} |

### 6.3.1 特性示例

6.3.1.1 调用非模板基类的成员函数

|  |
| --- |
| class Base{public:    void bFunc(){        cout << "Base::bFunc" << endl;    }};class Derived : public Base{public:    void dFunc(){        bFunc();    //调用基类Base的成员函数bFunc()        cout << "Derived::dFunc" << endl;    }};int main(){    Derived d;    d.dFunc();    return 0;} |

6.3.1.2 调用模板基类的成员函数

6.3.1.2.1 问题

|  |
| --- |
| template <class T>class Base{public:    void bFunc(){        cout << "Base::bFunc" << endl;    }};template <class T>class Derived : public Base<T>{public:    void dFunc(){        //error: there are no arguments to ‘bFunc’ that depend on a template parameter,        //so a declaration of ‘bFunc’ must be available        bFunc();    //调用基类Base的成员函数bFunc()        cout << "Derived::dFunc" << endl;    }};int main(){    Derived<int> d;    d.dFunc();    return 0;} |

6.3.1.2.2 三种解决方案

|  |
| --- |
| template <class T>class Base{public:    void bFunc(){        cout << "Base::bFunc" << endl;    }};template <class T>class Derived : public Base<T>{public:    void dFunc(){        this->bFunc();    //调用基类Base的成员函数bFunc()        cout << "Derived::dFunc" << endl;    }};int main(){    Derived<int> d;    d.dFunc();    return 0;} |

|  |
| --- |
| template <class T>class Base{public:    void bFunc(){        cout << "Base::bFunc" << endl;    }};template <class T>class Derived : public Base<T>{public:    void dFunc(){        Base<T>::bFunc();    //调用基类Base的成员函数bFunc()        cout << "Derived::dFunc" << endl;    }};int main(){    Derived<int> d;    d.dFunc();    return 0;} |

|  |
| --- |
| template <class T>class Base{public:    void bFunc(){        cout << "Base::bFunc" << endl;    }};template <class T>class Derived : public Base<T>{public:    using Base<T>::bFunc;    void dFunc(){        bFunc();    //调用基类Base的成员函数bFunc()        cout << "Derived::dFunc" << endl;    }};int main(){    Derived<int> d;    d.dFunc();    return 0;} |